Compile-time Detection of Machine Image Sniping

Martin Kellogg
Paul G. Allen School of Computer Science and Engineering
University of Washington
Seattle, USA
kelloggm@cs.washington.edu

***Abstract*—Machine image sniping is a difficult-to-detect security vulnerability in cloud computing code. When programmatically initializing a cloud computer, a developer specifies a machine image (file system and executables). The developer should restrict the search to only those machine images which their organization controls. Otherwise, the cloud computer might run a similarly-named malicious image. We present a lightweight type and effect system that detects requests to a cloud provider that are vulnerable to an image sniping attack, or proves that no vulnerable request exists in a codebase. We evaluated our type system on over 9 million lines of code, detecting 16 vulnerable requests with only 3 false positives.**

Machine image sniping is one example of the object construction problem: ensuring that objects are well-formed before they are used. Our type system generalizes to other instances of the object construction problem. In further experiments, it detected misuses of builders generated by the Lombok and AutoValue frameworks.

We further generalize our work to *accumulation analysis*, a special case of typestate analysis. Unlike arbitrary typestate analysis, accumulation analysis can be implemented soundly without a heavy-weight whole-program alias analysis. Our type system for detecting machine image sniping is an accumulation analysis.

***Index Terms*—pluggable types, AMI sniping, AWS, EC2, Java, lightweight verification, object construction, builder pattern, Lombok, AutoValue, accumulation analysis**

I. The AMI sniping problem

A client of a cloud services provider can create virtual computers programmatically, using the provider's public API. An *image* is the virtual computer's file system; it includes an operating system and additional installed software, and so it determines what code runs on the virtual computer.

For example, a client of Amazon Web Services indicates what image to use via the `DescribeImagesRequest` API (fig. 1). This API requires clients to carefully create requests to avoid a potential operational security risk [1].

There are three safe ways to select which image to use when sending a request to the API:

- Use the `setImageIds` method to specify a globally unique image ID.
- Use the `setFilters` method to set some criteria (such as the name of the image or its operating system), *and* use the `setOwners` method to restrict the images searched to those owned by the requester or some other trusted party.
- Use the `setFilters` method to set criteria that restrict the image to one that is owned by a trusted party using the "owner", "owner-id", "owner-alias", or "image-id" filters.

```
package com.amazonaws.services.ec2.model;

class DescribeImagesRequest {
  DescribeImagesRequest() {...}
  setOwners(String... owners) {...}
  setFilters(Filter... filters) {...}
  setImageIds(String... imageIds) {...}
}
```

Fig. 1: The `DescribeImagesRequest` API. A client constructs a `DescribeImagesRequest`, modifies it via the `set*` methods, then sends it to AWS to obtain a machine image.

```
request = new DescribeImagesRequest();
filter = new Filter("name", "RHEL-7.5_HVM_GA");
request.setFilters(filter);
api.describeImages(request);
```

Fig. 2: Client use of the DescribeImagesRequest API that is vulnerable to an AMI sniping attack.

The unsafe code in fig. 2 uses the "name" filter without an owner filter, which causes the API to return all the images that match the name. This introduces the potential for a so-called "AMI (Amazon Machine Image) sniping attack" [1], in which a malicious third party intentionally creates a new image whose name collides with the desired image, permitting the third party to surreptitiously inject their own code onto newly allocated machines. Any call that searches the public database without specifying some information that an adversary cannot fake is potentially vulnerable to a sniping attack and should be forbidden.

II. A type system for called methods

Our key insight is that **only certain combinations of method calls should be permitted**. We therefore designed a type system that records, for each object, the set of methods that have been invoked on that object. This information is represented in a *type qualifier* named `@CalledMethods`. A type qualifier is a modifier on a type, which makes the type more specific. Our implementation operates on Java programs, which represent type qualifiers via annotations, which start with `@`.

`@CalledMethods(`$methodList$`) Object obj` means that methods in $methodList$ have *definitely* been called on `obj`. For example, if `a()` and `b()` have been called on `obj`, then `obj` has type `@CalledMethods({"a", "b"})`. As additional methods are called using the same receiver expression, the type of that expression is refined to include more methods. Figure 3 shows part of the type hierarchy. The subtyping rule is:

$$\frac{A \supseteq B}{\texttt{@CalledMethods(A)} \sqsubseteq \texttt{@CalledMethods(B)}}$$

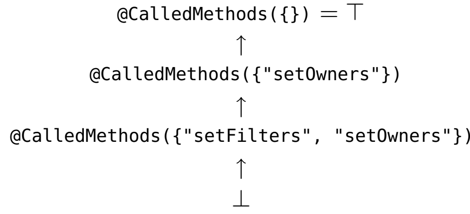

Fig. 3: Part of the type hierarchy for representing which methods have been called. If an expression's type has qualifier `@CalledMethods("setFilters", "setOwners")`, then the methods "`setFilters`" and "`setOwners`" have definitely been called on the expression's value. Arrows represent subtyping relationships. The diagram shows a part of the type hierarchy; the full hierarchy is a lattice of arbitrary size.

Our type system is flow-sensitive: a particular expression may have different types on different lines of the program, but must always be consistent with (i.e., a subtype of) the expression's declared type. Our type system relies on local type inference to compute updated expression types after method calls.

Though the type hierarchy has size up to 2^m where m is the number of methods in the program, the dataflow analysis (i.e., local type inference) is guaranteed to terminate: there are no unbounded ascending chains, which also means that there is no need to define widening operators (approximate $\sqcup$ operators).

We support a disjunctive type, `@CalledMethodsPredicate(P)`, which permits specifications like "`setOwners` $\vee$ `setImageIds`". The subtyping rules for disjunctive types are:

- `@CalledMethods(A)` $\sqsubseteq$ `@CalledMethodsPredicate(P)` if `A` makes `P` evaluate to `true`;
- `@CalledMethodsPredicate(P)` $\sqsubseteq$ `@CalledMethodsPredicate(Q)` if $\neg$(P$\Rightarrow$Q) is unsatisfiable;
- `@CalledMethodsPredicate(P)` $\sqsubseteq$ `@CalledMethods(A)` if $\neg$(P$\Rightarrow$Q) is unsatisfiable, where Q is the conjunction of the methods in A.

To detect and prevent erroneous calls, an API designer writes a specification—that is, they write types on formal parameters. For detecting AMI sniping, the corresponding specification is written on the parameter to the `describeImages` API in the AWS SDK (for presentation, the full specification has been shortened [2]):

```
DescribeImageResponse describeImages(
   @CalledMethodsPredicate("setImageIds || setOwners")
   DescribeImageRequest request);
```

Given this specification for `describeImages`, the typechecker rejects any call whose receiver has not had either `setImageIds` or `setOwners` called on it. This specification is sound: it prevents all AMI sniping attacks.

In general, the typechecker either:

- proves that a codebase only contains calls that are consistent with the specification, or
- issues an error, indicating possibly-defective code.

A. *Supporting type systems*

Our type system uses two supporting analyses to improve precision: a limited alias analysis that identifies methods that return their own receiver, and a constant propagation analysis.

TABLE I: Detection of AMI sniping vulnerabilities.

	Open source	Closed source
Projects	36	509
Non-comment non-blank lines of Java code	427K	8.7M
Manually-written annotations	5	29
True positives	3	13
False positives	2	1

In addition to what is shown in fig. 1, the `DescribeImagesRequest` class also defines *fluent* APIs: methods that return the same object they are called on. Fluent methods permit chained method calls:

```
describeImages(new DescribeImagesRequest()
                .withOwners(myOrg)
                .withFilters(filters));
```

To verify this code, it is necessary to know that each fluent setter method returns its receiver. To express this specification, we introduce a new type annotation: `@This`. When written on a method's return type, it indicates that the return value of the method is always exactly the receiver object (`this` in Java).

A common false positive we observed when applying only the `@CalledMethods` type system to code that calls the `describeImages` API is that it is also possible to specify an owner using a particular filter, without actually calling `withOwners()`. We plugged the Checker Framework's constant propagation analysis [3] into the `@CalledMethods` type system to eliminate these false positives, by treating calls that set an owner via a filter the same as direct calls to `withOwners()`.

B. *Implementation*

We implemented this type system for Java using the Checker Framework [4]. The framework provides local type inference, among other conveniences, which means that programmers only need write type qualifiers on formal parameters, fields, and return types. However, because most types are inferred, even these are rarely needed. Our tool, called the Object Construction Checker, is available at https://github.com/kelloggm/object-construction-checker.

III. Evaluation

We evaluated our approach to detecting AMI sniping attacks on two corpora of codebases:

- 36 open-source codebases: non-duplicate projects from GitHub that use the `describeImages` API and build under Java 8 using Gradle or Maven.
- 509 codebases from an anonymous industrial partner, each of which contains calls to the `describeImages` API.

The results appear in table I. Our tool found 16 codebases potentially vulnerable to third-party abuse via AMI sniping. The closed-source developers fixed each potential vulnerability. An example true positive, from Netflix/SimianArmy, appears in fig. 4. If the list of image ids is null, then the code fetches every AMI available. Though the method's documentation does not say so, it is incumbent on any caller of this code to filter the result after the fact.

```
DescribeImagesRequest request = new DescribeImagesRequest();
if (imageIds != null) {
    request.setImageIds(Arrays.asList(imageIds));
}
DescribeImagesResult result = ec2Client.describeImages(request);
```

Fig. 4: A true positive AMI sniping concern in Netflix's Simian-Army project.

A false positive is a case where our type system could not verify safe code. Our tool reported 3 false positives (precision = 84%), due to projects that wrap the `describeImages` API with methods that take a list of `Filter` objects. Our type system cannot express that a list of `Filter` objects must contain the correct filters.

The tool required one annotation per 268,000 lines of code. We wrote these on helper methods that wrap setter calls. It was straightforward to write these annotations, because our tool issues an error wherever an annotation is required.

IV. The object construction problem

The AMI sniping problem is one instance of the more general *object construction problem* of ensuring that objects are well-formed before they are used.

The standard API for Java object construction contains one constructor for each combination of possible values that results in a well-formed object. Alternate patterns for object construction have been devised, such as the *builder pattern* [5]. To use the builder pattern, the programmer creates a separate "builder" class, which has two kinds of methods:

- *setters*, each of which provides a *logical argument*—a value that ordinarily would be a constructor argument
- a *finalizer* (often named `build`), which actually constructs the object and initializes its fields appropriately.

The builder pattern improves readability and flexibility in client code, but it loses compile-time verification that logical arguments are provided.

`DescribeImagesRequest` is a builder: the `with*` and `set*` methods are setters and `describeImages()` is the finalizer. The compiler permits all combinations of method calls, so a client can accidentally fail to set the owner when setting the name.

V. Lombok and AutoValue builders

Lombok [6] and AutoValue [7] are widely-used Java code generation libraries that allow developers to avoid writing boilerplate code. Both libraries provide an `@Builder` annotation [8], [9] that developers can write on class C to generate a builder class for C. To use the builder class, a client creates a builder object, incrementally adds information to it by calling setter methods corresponding to C's fields, and then calls the finalizer method `build()` to construct a C object. If some fields of C have types that are annotated as `@NonNull`, then `build()` throws a null-pointer exception if any such field has not been set, crashing the program. These crashes frustrate developers.

For example, consider an application developer who depends on a library like Yubico/java-webauthn-server, which includes the class in fig. 5. Figure 6 is an example of such code, from java-webauthn-server's included demo. As defined, this code

```
@Builder
public class UserIdentity {
    private final @NonNull String name;
    private final @NonNull String displayName;
    private final @NonNull ByteArray id;
}
```

Fig. 5: A class that has a builder. The `@Builder` annotation causes Lombok to generate a builder at compile time. This example is simplified code from the Yubico/java-webauthn-server project.

```
UserIdentity.builder().name(username).displayName(displayName)
    .id(generateRandom(32)).build()
```

Fig. 6: A client of the `UserIdentity` builder in fig. 5, from the same project. This builder use will not cause a run-time exception, because all fields whose type is `@NonNull` have been set.

works correctly. However, suppose that a developer of java-webauthn-server adds another field to `UserIdentity`. If this field's type is annotated as `@NonNull`, then the code in fig. 6 will begin to fail—at run time!—when the library dependency is updated. Even if this is caught during testing, debugging the cause can still be painful because the bug will manifest as a null-pointer exception in the unmodified client code.

These sorts of bugs could be avoided by checking—at compile time—that the setter for each field whose type is non-null has been called before `build` is called. Clients prefer compile-time checking that mandatory fields are set on builders; it is one of Lombok's most requested features [10]–[17].

A. Handling Lombok and AutoValue builders

The Object Construction Checker can infer most specifications for setter and finalizer methods generated by Lombok and AutoValue, so programmers do not need to write them.

It adds an `@This` annotation to return types of setter methods in Lombok and AutoValue builders. It adds an `@CalledMethods` annotation to finalizer methods generated by Lombok and AutoValue. For Lombok, the annotation argument contains the setters for fields annotated as `@NonNull`, except fields annotated as `@Singular` or `@Builder.Default`. For AutoValue, the annotation argument contains the setters for each field which is not nullable, `Optional`, or a Guava Immutable type.

B. Evaluation on Lombok and AutoValue builders

We performed case studies and a small user study.

The case studies (table II) demonstrate *the costs and benefits of onboarding existing projects to use our tool*. We chose 5 popular projects from GitHub with significant builder usage that compiled with our infrastructure. We were not familiar with the projects or their use of Lombok or AutoValue. We found a bug in gapic-generator which we reported to the developers, who promptly verified and fixed the issue, saying "your static analysis tool sounds truly amazing!" [18] We also removed a complex type hierarchy in java-webauthn-server that forced clients to to call setters in a pre-defined order, to ensure that all mandatory fields were set. Our analysis made it redundant.

The user study shows the *ongoing benefits of using the tool*. Each of our six participants was employed as a software engineer (all at the same level within their company but on different teams), regularly uses Java, and was familiar with Lombok. Participants were not familiar with our tool. We chose

TABLE II: Verifying uses of the builder pattern. Throughout, "LoC" is lines of non-comment, non-blank Java code. "Annos." is number of manually-written annotations to specify existing methods. "TPs" is true positives. "FPs" is false positives, where the Object Construction Checker could not prove that the call was safe, but manual analysis revealed that it could not fail at run-time.

Project	Framework	LoC	Finalizer calls	LoC added	LoC removed	Annos.	TPs	FPs
Yubico/java-webauthn-server	Lombok	7,153	42	52	426	48	0	3
javagurulv/clientManagementSystem	Lombok	5,134	65	0	0	0	0	0
google/error-prone	AutoValue	74,180	9	0	0	2	0	2
googleapis/gapic-generator	AutoValue	49,054	442	2	0	58	1	1
google/nomulus	AutoValue	71,627	95	0	0	23	0	8

two different classes for participants to add a new field to. One class had a test case written in Java; the other class had no test. We used a factorial design: each participant executed the task for both of these classes; for one, they had access to our tool, and for the other, they did not.

3/6 participants failed to complete the task without our tool (two in the condition lacking a failing test), but all 6 succeeded with our tool. There was a difference in means in the time taken when considering only those who finished both tasks: using our tool was 1.5x faster (≈200 vs. ≈306 seconds).

5/6 participants said they encountered tasks like these at least monthly. The subjects were also convinced that the compile-time warnings were useful. For example, one subject said "It was easier to have the tool report issues at compile time." Several also mentioned the tool's value in localizing where to make changes: for example, one said the tool "allowed me to immediately hone in on the problem."

VI. Accumulation analysis

We define *accumulation analysis* as a special case of typestate analysis [19] that can be performed modularly without an alias analysis.

A typestate system permits an object's type to change as a result of operations in the program. For example, in a typestate system, a file's type might change from `UnopenedFile` to `OpenedFile` to `ClosedFile`. Operations like `read()` are permitted only on an `OpenedFile`.An arbitrary typestate analysis requires alias analysis for soundness. Suppose that two `OpenedFile` references `f1` and `f2` might refer to the same file object. Calling `f1.close()` must change the estimate of the type of `f2`, or else the analysis would permit the program to perform the possibly illegal operation `f2.read()`.

An accumulation analysis is a program analysis where the analysis abstraction is a monotonically increasing set, and an operation is legal only when the set is large enough—that is, the estimate has accumulated sufficiently many items. Accumulation analysis is a special case of typestate analysis in which (1) the order in which operations are performed does not affect what is subsequently legal, and (2) the accumulation does not add restrictions; that is, as more operations are performed, more operations become legal.

For builders, each typestate stands for a different set of logical arguments that have been provided. The finalizer operation is permitted in all typestates whose set is a superset of the required logical arguments. Verifying builder usage—for preventing AMI sniping or null-pointer exceptions in Lombok or AutoValue builders—is an accumulation analysis.

We believe that many typestate properties can actually be expressed as accumulation analyses. `File` or `Socket` objects, e.g., are rarely closed and re-opened: new objects are created instead. Many real-world problems are of the simple form "Always call m before n," involving a single operator (e.g., requiring a call to an initializer method).

An accumulation analysis is free to only do partial reasoning about aliasing, or no reasoning at all. Ignored aliases can cause imprecision and false positive warnings, but never unsoundness. Suppose that `a1` and `a2` are must-aliased, and their estimate of logical arguments supplied is $\{x, y\}$. The operation `a1.z()` changes `a1`'s estimate to $\{x, y, z\}$. The valid operations on the old type are a subset of the valid operations on the new type.

VII. Related Work

No other static analysis exists which detects an image sniping attack. AWS acknowledged the vulnerability, but cannot revoke this widely-used API nor change its behavior incompatibly. Their proposed mitigation is "to follow the best practice and specify an owner" [20]. An independent security researcher published instructions to detect if running virtual machines are impacted, but said that following best practices is the best mitigation [21]. A sound static analysis like ours does not depend on programmers to remember to use the best practice.

The object construction problem motivates some language design choices such as named and default parameters in languages like Python. The AutoValue Step builder [22] and the Jilt library [23] generate builders that require clients to set mandatory fields in a pre-defined order before setting any optional fields. Our analysis is applicable to legacy code and does not prescribe an order for setter calls.

Heap-monotonic typestates [24] are those in which "statically observable object invariants only become stronger as objects evolve." Like accumulation analysis, it can be verified without alias analysis. It focuses on data structure invariants but cannot check correctness of client code. It cannot express properties where methods may be invoked in an arbitrary order, like the required methods property for builders. Another difference from our work is that heap-monotonic typestate was never implemented nor evaluated.

An extended version of this paper is available [2].

Acknowledgments

Thanks to my collaborators Manu Sridharan, Manli Ran, and Michael D. Ernst.

References

[1] MITRE, "CVE-2018-15869," https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-15869, 2018.

[2] M. Kellogg, M. Ran, M. Sridharan, M. Schäf, and M. D. Ernst, "Verifying object construction," in *ICSE 2020, Proceedings of the 42nd International Conference on Software Engineering*, Seoul, Korea, May 2020.

[3] *The Checker Framework Manual: Custom pluggable types for Java*, http://CheckerFramework.org/.

[4] M. M. Papi, M. Ali, T. L. Correa Jr., J. H. Perkins, and M. D. Ernst, "Practical pluggable types for Java," in *ISSTA 2008, Proceedings of the 2008 International Symposium on Software Testing and Analysis*, Seattle, WA, USA, July 2008, pp. 201–212.

[5] E. Gamma, R. Helm, R. E. Johnson, and J. Vlissides, *Design Patterns*. Reading, MA: Addison-Wesley, 1995.

[6] R. Zwitserloot and R. Spilker, "Project Lombok," https://projectlombok.org/, 2019, accessed 19 April 2019.

[7] K. Bourrillion and É. McManus, "AutoValue," https://github.com/google/auto/tree/master/value, 2019, accessed 14 August 2019.

[8] The Lombok Authors, "@Builder," https://projectlombok.org/features/Builder, 2019, accessed 12 February 2019.

[9] K. Bourrillion and É. McManus, "AutoValue with builders," https://github.com/google/auto/blob/master/value/userguide/builders.md, 2019, accessed 14 August 2019.

[10] jax, "Required arguments with a Lombok @Builder," https://stackoverflow.com/questions/29885428/required-arguments-with-a-lombok-builder, 2015, accessed 20 August 2019.

[11] A. Kamangir, "Using Lombok to create builders for classes with required and optional attributes," https://stackoverflow.com/questions/54155315/using-lombok-to-create-builders-for-classes-with-required-and-optional-attribute, 2019, accessed 20 August 2019.

[12] M. Punjabi, "FindBugs detecter for NonNull Lombok builder attributes," https://stackoverflow.com/questions/51324922/findbugs-detecter-for-nonnull-lombok-builder-attributes, 2018, accessed 20 August 2019.

[13] C. Brunotte, "Mark fields as required for Builder," https://github.com/rzwitserloot/lombok/issues/1043, 2016, accessed 20 August 2019.

[14] B. Lynch, "[FEATURE] @StepBuilder," https://github.com/rzwitserloot/lombok/issues/2055, 2019, accessed 20 August 2019.

[15] A. Nakagawa, "Feature: Allow fields to be specified only via builder's constructor," https://github.com/rzwitserloot/lombok/issues/1303, 2017, accessed 20 August 2019.

[16] F. Friis, "Calling final builder step without providing required arguments," https://github.com/rzwitserloot/lombok/issues/1202, 2016, accessed 20 August 2019.

[17] C. Beams, "@Builder should require invoking methods associated with final fields," https://github.com/rzwitserloot/lombok/issues/707, 2014, accessed 20 August 2019.

[18] M. Sridharan, "Possible missing packageinfo property in javasurface-transformer," https://github.com/googleapis/gapic-generator/issues/2892, July 2019.

[19] R. E. Strom and S. Yemini, "Typestate: A programming language concept for enhancing software reliability," *IEEE Transactions on Software Engineering*, vol. SE-12, no. 1, pp. 157–171, January 1986.

[20] J. Bicha and N. Alvine, "CVE-2018-15869: –owners flag isn't mandatory," https://github.com/aws/aws-cli/issues/3629, 2018, accessed 5 June 2019.

[21] S. Piper, "Investigating malicious AMIs," https://summitroute.com/blog/2018/09/24/investigating_malicious_amis/, 2018, accessed 5 June 2019.

[22] K. Sopko, "auto-value-step-builder," https://github.com/sopak/auto-value-step-builder, 2019, accessed 14 August 2019.

[23] A. Ruka, "The type-safe builder pattern in java, and the jilt library," https://www.endoflineblog.com/type-safe-builder-pattern-in-java-and-the-jilt-library, 2017, accessed 15 August 2019.

[24] M. Fähndrich and K. R. M. Leino, "Heap monotonic typestates," in *IWACO 2003: International Workshop on Aliasing, Confinement and Ownership in object-oriented programming*, Darmstadt, Germany, July 2003.